

**APPLICATION
FOR
UNITED STATES LETTERS PATENT**

APPLICANT NAME: Stephen E. Fischer
TITLE: Dependency Specification Using Target Patterns
DOCKET NO.: FIS920000280US1

INTERNATIONAL BUSINESS MACHINES CORPORATION
NEW ORCHARD ROAD, ARMONK, NY 10504

CERTIFICATE OF MAILING UNDER 37 CFR 1.10

I hereby certify that, on the date shown below, this correspondence is being deposited with the United States Postal Service in an envelope addressed to the Assistant Commissioner for Patents, Washington, D.C., 20231 as "Express Mail Post Office to Addressee"

Mailing Label No. EK140407722US

on 1/10/01

K. C. INQ - MARS

Name of person mailing paper

Karen King-Mars
Signature

1/10/01
Date

IBMF100320000

Dependency Specification Using Target Patterns

Background Of The Invention

1. Field of the Invention

This invention relates to a method for updating of source code and object code
5 in a computer program and, more particularly, a software utility program for
automating selective updates of source code and corresponding object code.

2. Description of Related Art

The relevant prior art, upon which the present invention includes improvements to, is
10 the *make* utility originally developed to run on UNIX® systems. A brief discussion of
the *make* command or utility in a UNIX environment is herein discussed to serve as
background in disclosing the present invention. The *make* utility is a software
engineering tool for managing and maintaining computer programs. Typically, a
program may consists of many component files, and as the number of files in the
program increases, the compile time also increases. Similarly, the complexity of
15 compilation commands can increase with the program complexity. In the same way,
the likelihood of human error in executing many compilation commands and updating
source files can also increases with the program complexity.

The *make* utility uses a descriptor file containing dependency rules, macros,
and suffix rules to instruct *make* to automatically rebuild the program whenever one
20 of the program component files is modified. The *make* utility operates by following
rules that are provided in its descriptor file, typically called *makefile*. The *make* utility
saves compile time by selectively recompiling only the files that were effected by
changes. Thus, the *make* utility simplifies the problem of keeping track of modified
files, recompiling files, and re-linking those files to produce an executable program.

25 However, the *make* command is usually written using shell commands or *make*
syntax. This does not include other languages that may offer greater flexibility for the
programmer and greater flexibility in operation of the utility.

The *make* file includes two elements: targets, and dependencies. The *make* utility works by comparing the time stamp of each target and its dependencies. If at least one prerequisite file (such as a source file) is newer than the associated target file (such as an associated object file), the *make* file executes a typical rule, for example, recompiling a source file.

The *make* utility is mostly used to sort out dependency relations among files. Generally, even relatively small software projects involve a number of files that depend upon each other. For example, a program must be linked from object files, which in turn must be created from assembly language or high-level language source files. If one or more source files are modified, the program must be re-linked after recompiling some, but not necessarily all, of the sources. This selective building is normally repeated many times during the course of the project. It is this process that the *make* utility addresses. However, the *make* utility requires explicitly detailing the source files or target files needing to be changed or updated which can take significant amounts of time and be cumbersome. Also, the *make* command requires building a substantially complete dependency tree before it starts, which can also take an extensive amount of time and computer resources.

Typically, a programmer records all the relationships among set files, and then uses the *make* utility to automatically perform all updating tasks. The *make* command has this general form: "*make myprog*". The *make* utility then carries out only those tasks necessitated by source file changes since the previous *make* command. For example, the *make* utility examines the file system to determine when the relevant files were last modified. If file A depends on file B, and if B was modified after A, then A must be remade-compiled, linked, edited, substituted in a library.

The command, "*make myprog*" (where "myprog" is a target program), indicates that the programmer wants to update the version of the program, named "myprog". Thus, if "myprog" is an executable file, the command indicates the desire to perform all necessary compilation and linking required to create the file. Instead of entering a great many compiler commands, the *make* utility automates the process.

The program requiring an update ("myprog" in the above example) is called the target of the operation. The program is built from one or more associated files called prerequisites or dependents. Each of these files, may in turn, have other files as prerequisites or dependents. For instance, executable programs are built by linking object files. When source files have changed, typically it is desirable to recompile the object files before linking. Thus, each source file is the prerequisite for each object file.

The *make* utility is sensitive to hierarchies and dependencies, such as source to object, and object to executable relationships. The programmer is responsible for specifying some dependencies in a description file, however, *make* can determine many dependencies for itself. In deciding which files to build and how to build them, *make* draws on the names of existing files, the last modified kinds of the files, and a set of built in rules. With this background, a command, like, "*make myprog*", operates such that all necessary parts of the hierarchy are updated.

Thus, the *make* utility was created to help programmers manage software development projects and files associated with those projects. The need for such a tool as the *make* command stems from typical scenarios encountered by programmers and software developers. For example, if a programmer is working on a number of programs that will be compiled and linked together to form a single executable program, every time a file is changed the file must be recompiled, then re-linked with the existing object file to create a new executable program.

This sequence presents a significant problem in which the programmer has to remember which files have been modified and, consequently, which files have to be recompiled. Thus, recompiling soon becomes a cumbersome and a difficult task prone to errors and omissions. The *make* utility handles many details in compiling modified files. To accomplish the task, *make* follows a set of rules that are provided in a special file called a *makefile*.

More specifically, *make* operates by comparing the date and time of a source file with the date and time of the associated object file. Together, the date and time

are called the time stamp. If the comparison shows that the source file is newer than the object file, or if the object file does not exist, *make* performs the task listed in the *make* file to convert the source file into an object file. In contrast, if the object file is newer than the source file, then *make* recognizes that it does not have to recompile
5 the source file.

The *make* utility requires that a target/dependency line must begin in the first column of the line. A command line must be indented. A target is a name followed by a colon. The name appears at the beginning of the line. Two types of dependencies occur in a *make* file, direct dependencies and indirect dependencies. A
10 direct dependency appears with the target. This means that the target depends on the file or files listed after the colon. An indirect dependency is indirectly related to the first target. However, typically, multiple *makefiles* are needed to specify cross-directory file dependencies.

Bearing in mind the problems and deficiencies of the prior art, it is therefore
15 an object of the present invention to provide a utility program which more rapidly and efficiently updates target files and their corresponding dependent files.

It is another object of the present invention to provide a utility program using a
more flexible programming language.

A further object of the invention is to provide a more effective search
20 technique requiring less intensive computer resources and increased efficiency in the searching process.

It is yet another object of the present invention to provide a utility program which more effectively finds target files.

Still other objects and advantages of the invention will in part be obvious and
25 will in part be apparent from the specification.

Summary of the Invention

The above and other objects and advantages, which will be apparent to one of skill in the art, are achieved in the present invention which is directed to, in a first aspect, a method for updating existing code in a computer program after inputting new code which defines changes to the existing code, wherein the method comprises generating a target file list which includes target files, generating an associated file list including associated files which correspond with the target files, and executing an algorithm. The algorithm locates the target files by employing a search process. The algorithm updates the target files and updates the associated files by selectively compiling the target files.

In a related aspect, the present invention provides source code and object code. The target files are source code and the associated files are object code. The source code are selectively compiled to update and provide the associated object code.

In another related aspect, the present invention provides updating the associated file list with new associated files. The new associated files are defined by the new code.

In yet another related aspect, the present invention provides a search technique including pattern type variables which use generic rules to specify the associated object code for updating.

In another aspect, the present invention provides a method for generating changes and updating existing files and code in a computer program wherein the method comprises reading existing source code and existing object code in the computer program. Then the method reads a plurality of associated files, where the associated files are associated with the source code, and executes a utility program which updates target source code and the object code associated with the target source code. The utility program includes scripting language specifying particular characters to search for in the target code and the associated code. The method further generates a target code list for the source code and the associated object code by using the utility program. The method then updates the target code and the associated code by

employing a search technique specified in the utility program. The search technique includes specifying patterns such that the specified patterns identify the existing associated code being updated.

In a related aspect, the present invention provides a search technique including
5 matching specified characters in the target code and the associated code such that the specified characters identify the existing associated code being updated.

A related aspect of the present invention provides including specified patterns of the search technique including pattern type variables which use generic rules to specify the associated object code for updating.

10 In a related aspect, the present invention provides a search technique including pattern type variables which use generic rules to specify the associated object code for updating.

In a further aspect, the present invention provides a method for generating changes and updating existing source code and existing object code in a computer
15 program which includes a plurality of dependent files which are prerequisites of the source files, wherein the method comprises inputting a utility program which updates the target source code and the object code associated with the target source code. The utility program includes a scripting language specifying particular characters to search for in the target code and the associated object code. Further, the method generates a
20 target code list for changing the associated object code using the utility program, and executes a search technique including pattern type variables which use generic rules to specify the associated object code to be changed, and then updates the associated object code.

In a related aspect, the present invention provides a utility program defines
25 new source code to be added to the existing source code.

In another related aspect, the present invention provides a utility program
prioritizes the target code to update while employing the search technique.

s/f³
In yet another related aspect, the present invention provides a utility program includes a process procedure for an operator to call, the process procedure recursively invokes the utility program and arguments.

In still another related aspect, the present invention provides a utility program
5 in a UNIX® environment.

s/f⁴
In a related aspect, the present invention provides a utility program provides that the existing code with a specific character are not considered to be a file, and thereby are bypassed for any changes by the utility program.

In another aspect, the present invention provides a computer program product
10 for updating existing code in a computer program after inputting new code which defines changes to the existing code, wherein the computer program product includes computer readable program code means for generating a target file list which includes target files. The computer program further includes computer readable program code means for generating an associated file list including associated files which correspond
15 with the target files. A computer readable program code means for executing an algorithm is also included where the algorithm locates the target files by employing a search process, the algorithm updates the target files and updates the associated files by selectively compiling the target files.

[] In yet another aspect, the present invention provides a program storage device
20 readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for updating existing code in a computer program after inputting new code which defines changes to the existing code, wherein the method includes generating a target file list which includes target files. Then, the method generates an associated file list which includes associated files which correspond with the target files, and executes an algorithm which locates the target
25 files by employing a search process. The algorithm updates the target files and updates the associated files by selectively compiling the target files.
[]

Brief Description of the Drawings

The features of the invention believed to be novel and the elements characteristic of the invention are set forth with particularity in the appended claims. The figures are for illustration purposes only and are not drawn to scale. The 5 invention itself, however, both as to organization and method of operation, may best be understood by reference to the detailed description which follows taken in conjunction with the accompanying drawings in which:

Fig. 1 is a top level flow chart of a preferred embodiment of the present invention.

10 Fig. 2 is flow chart of a sub-process of the embodiment of the present invention shown in Fig. 1.

Fig. 3 is a flow chart of another sub-process of the embodiment of the present invention shown in Fig. 1.

15 Fig. 4 is a schematic of a computer for running a computer program embodying the method of the present invention.

Description of the Preferred Embodiment(s)

In describing the preferred embodiment of the present invention, reference will be made herein to Figs. 1-3 of the drawings in which like numerals refer to like 20 features of the invention. Features of the invention are not necessarily shown to scale in the drawings.

25 The present invention discloses using target patterns to provide a concise way to specify entire classes of dependencies. This method can replace many concrete (non-pattern-specific) rules in a build control file such as the *makefile* disclosed in the prior art. Other prior art methods use algorithms to build a complete concrete dependency tree before the utility starts. The present invention builds a pattern-based dependency tree instead, which can be done in minimal time without operating system

slf/c

calls. Further, the present invention determines which files match the patterns as needed, thus saving time and computer resources.

Referring to Fig. 1, a top-level flow chart is shown disclosing the present invention 10. A preferred embodiment of the present invention is implemented as a utility program called *updt*. The top-level flow chart of the *updt* utility program includes five steps 12, 14, 16, 18, and 20. The first step 12 starts the program by entering a command at the command line. The next step 14, includes processing the command line arguments and generating a target list of source files or source code. Next, in step 16, the program reads a dependency file list including object code into the global update rules and in prerequisite arrays. Thus, the program is first retrieving the target list of files or code to update, and then reading what files or code are associated with, or dependent on, that target list of files or code. Then, in step 18, the program applies the process procedure to the target list of object code. The process procedure then updates all target files or code, and the program stops, step 20.

Referring to Fig. 2, a flow chart is shown disclosing the dependency file read sub-process 50 of the present invention shown in step 16 of Fig. 1. The process starts 52 by having a dependency file to read. Next, each line in the dependency file is read, step 54, by proceeding to a series of "if" statements, steps 56, 58. Then, in step 56, for each line in the dependency file the program ascertains whether the line starts with a pound (#) sign or is blank 56. If the line does not start with a pound sign or is blank this statement is false, and the program continues to look if the line is indented 58, if this is false the program splits the line at the colon (:) mark into target and prerequisite strings.

If at step 56 the line does start with a # or is blank, then the "if" statement at step 56 is true, and the program proceeds to step 70. If at step 58 the line is indented, then the "if" statement is true, and the most recently unentered target is appended to the rule list (if not already there), and the line is appended to corresponding rule array entry 59. Then the program continues to step 70.

After step 60, the program performs Tcl substitutions in a target string 62. Tcl is the preferred programming language in the present embodiment described herein, however, other interpreted programming languages may be used, such as, perl, or python®. Then, the path abbreviations are expanded in the target and the 5 prerequisite strings 64. Next, the target is appended to the requisition list 66. Then, prerequisites are stored in corresponding requisite array entries 68. Once this is completed, the program ascertains whether this is the last line 70, if it is the last line the program stops 72, if it is not the last line the program returns to another line in the dependency file, step 54.

10 Referring to Figure 3, a flow chart is shown disclosing the "process procedure" sub-process 100 of the present invention shown in step 18 of Fig. 1. The process starts with the start command having a target 102. For each specified target in the target list find a first match in the requisition list target then get the prerequisite list from corresponding requisite array 106. Next, do all possible Tcl substitutions in 15 the prerequisite lists, including the target for \$trg 108. Then, for each prerequisite recursively process the prerequisite as a new target 110. Next, complete all the TCL substitutions on the target including \$trg(s) and \$src(s) . Then ask if the target is concrete 114, if this is true, ask if the target is older than any prerequisite 116. If this is true, execute the rule array member associated with this target to generate the 20 target 118. After this, ask if this is the last target 120, if this is true, stop 122, if it is false, go back to the beginning 104 and ask for another specified target in the target list.

If the answer to step 114 is false, that is, the target is not concrete, then the process is to go to step 120 to ask if this is the last target. Also, if the answer to step 25 116 is false, that is the target is not older than any prerequisite, then the process goes to step 120 to ask if this is the last target.

A further description of the present invention, being deployed, is explained below, referring to Figs. 1, 2, and 3. As shown in Fig. 1, a top level flow chart 10 is shown outlining the basic steps of the present invention. The method of the present

invention starts with processing command line arguments to get a target list, step 14. The user types a command (in this preferred embodiment *upd1*) followed by one or more "targets". The computer system then processes the command line instruction, step 14. The preferred embodiment disclosed herein uses a Unix® platform, however other platforms may be used, such as, Microsoft Windows®. The program then processes the command and the target list.

Next, the dependency file is read into the global update rule and prerequisite arrays, step 16. The dependency file describes which files each valid target depends on and also how to generate each target from those files. The dependency file lists the dependent files on the target list. Any target file without an explicit or implicit (/) in its name is assumed to be abstract, i.e., that is, it is not considered to be a file. Target patterns are used to support generic dependency specifications and generic update rules.

Targets may be specified with either glob-type patterns using the standard shell wild cards, or regular expression patterns. This generalization permits the concise specification of entire families of dependencies and/or update rules. This facility is in contrast to prior art (for example, *make*'s suffix rule concept). Further, the dependencies and update rules for a given target need not be specified together. Then, the program applies the process procedure to the target list. Thereafter the program stops 20.

Referring to Fig. 2 the use of the "dependency file read" is further explained. The method for the "dependency file read" starts with each line in the dependency file being read, step 54. The program then asks the question if the line starts with # or is blank, step 56. If that statement is true, the program proceeds to ask whether this is the last line 70, if it is the last line the program stops 72. If it is not the last line the program reads the next line in the dependency file, step 54. After the "if" statement the program proceeds to another "if" statement, asking whether the line is indented, step 58. If this statement is true, the program appends the last target to the rule list, step 59, if not already there. It also appends the line to the corresponding rule array

entry, step 59. If the last "if" statement is not true, the program splits the line at the colon (:) mark, into target and prerequisite strings, step 60.

Next, the program performs Tcl substitutions in target string, step 62. Tokens in a target or prerequisite list which occur before the first token with a slash (\) in it 5 are considered to be abstract. Abstract prerequisites trigger the rule in which they occur as target as though they were out of date files (whether they actually exist as files or not).

The program then expands path abbreviations in target and prerequisite strings, step 64. Then the program appends target to the requisition list, step 66. Next, the 10 program stores prerequisites in corresponding requisition array entry, step 68. Then, the program will ask if this is the last line of the file, step 70, if it is, the program stops, step 72, if it is not the last line, the program proceeds to read the next dependency file, step 54.

Referring to Fig. 3, the use of the "process procedure" is further explained. 15 The program reads each target in the target list, step 104. Then, for each specified target in the target list the program first finds a first match in the requisite list to the target, step 106, and retrieves the prerequisite list from the corresponding requisite array, step 106. Then, do all possible Tcl substitutions in the prerequisite list, including target for \$trg, step 108. Next, for each prerequisite recursively process the 20 prerequisite as a new target, step 110. Then, do all Tcl substitutions on the target, including \$trg(s) and \$src(s), step 112. Proceed to the "if" command, which reads, is the target concrete, step 114. If this is false, the program asks if this is the last target, step 120. If this last "if" statement is true, the program stops, step 122, if the "if" statement is false the program looks for matches for the next specified target in 25 the target list, beginning with step 104.

If the "if" statement 114, "is the target concrete", is true, the program continues to another "if" statement 116. This "if" statement asks if the target is older than any prerequisite. The program is ascertaining if the target file needs to be including in a recompiling process. If any of the prerequisite files of that target file

are newer than the target file, then the dependent file has been more recently updated, and the target file needs to be regenerated. Thus, if the answer is false, the program continues to the final "if" statement which asks is this the last target, step 120. If this is true it stops, step 122, if this is false, the program continues to look for the next specified target and a match for it, beginning with step 104. If the "if" statement 5 116, "is the target older than any prerequisite", is true, the program continues to execute the rule array member associated with this target to generate the target, step 118. Then, the program continues to the last "if" statement which asks if this is the last target, if this is true the program stops, step 122, if this is false, the program 10 looks for the next specified target in the target list, beginning with step 104.

Examples of dependencies and update rules and other aspects of the present invention are explained below, using the preferred embodiment utility called *upd*, which provides further illustration and clarification of the present invention. Any of the more powerful scripting languages (for example, perl, python®, Tcl) could be used 15 as the internal rule language. Also, the same scripting languages, or a compiled language such as, C or C++, could be used as implementation language.

The Tcl variables are set with "set variable-name variable-value" and references with "\$variable-name". Also, in-line code evaluation is triggered with square brackets. Shell command quoting rules are determined by the Tcl 20 environment, and are not the same as in a shell environment.

Updt is designed to support multi-directory builds from a single control file. For example, an *upd* rule can take the form:

```
hello.o: ./ hello.c  
        exec cc hello.c
```

25 The exec keyword is needed by Tcl for executing shell commands, and the './' before 'hello.c', tells *upd* to look in the current directory for that file. *Updt* relies on the variables in its underlying scripting language, preferably Tcl. These variables may be initialized as they would be in any Tcl script, e.g:

```
set CDIR [pwd] /src
```

sets \$CDIR to the 'src' subdirectory of the present working directory, and

```
    set SRCS ""  
    foreach f [glob -noncomplain $CDIR/* .c] {lappend SRCS $f}
```

initializes \$SRCS to all the *.c files in the 'src' subdirectory.

5 Several variables are set automatically by *updtk*. These include \$1-\$9 (parenthesized pattern sub-expressions), \$trg (active target), \$req (single prerequisite), \$sreqs (changed prerequisite), and \$reqs (all prerequisite). More variables can be devised.

Tcl procedures may be defined to improve the control file readability. For
10 example:

```
proc filePathList {dir {patn *}} {  
    # Generate list of file fullpaths in $dir matching pattern $patn  
    # $patn defaults to '*'.  
    set out ""  
15    foreach f [glob -nocomplain $dir/$patn] {lappend out $f}  
    return $out
```

With this procedure defined in the *Updtfile*, the SRCS variable assignment in the previous example can be re-written as:

```
set SRCS [filePathList $CDIR]
```

20 This kind of embedded scripting language support is powerful, and can improve control file clarity.

Tokens in a target or prerequisite list which occur before the first token with a "/" in it are considered to be abstract. Abstract prerequisites simply trigger the rule(s) in which they occur as target(s) as though they were out-of-date files (whether 25 they actually exist as files or not).

The first token in the prerequisite (or target) list with a forward "/" slash in it sets the initial directory context for the rest of the prerequisites (or targets). The forward "./" slash sets the context to the current directory (the directory in which *updtk* was invoked). This directory context can be dynamically changed, within the

list, using additional path switches. For example, if the "*.o" files in the "obj" subdirectory, and the "*.a" libraries in the "lib" subdirectory. A typical library dependency might then look like:

```
./lib/libxx.a: obj/ f1.o f2.o f3.o
```

- 5 The trailing slash in the "obj/" token makes it a path switch, rather than a prerequisite. All subsequent prerequisites are considered to be prefixed with "(pwd)/obj/".

The path switch facility is important for concise specification of cross-directory dependencies. Path switches beginning with "/" are considered to be absolute path
10 switches, completely overriding any prior path context. Other path switches are taken to be relative to the current path context. If the first path switch is relative, it is taken as relative to the current directory.

In summary, every target or prerequisite list which is intended to reference real files must initialize the directory context with a path switch before it lists those
15 files. This behavior is appropriate for a tool designed for multi-directory builds from a single control file.

The *updt* utility relies on the use of target patterns. A sample rule in *Updtfile* can look like:

```
set patternType glob  
20      (*) .o: $1.c  
          exec $CC $CFLAGS -c $req
```

The pattern type variable tells *updt* to interpret subsequent targets as glob
25 (shell-like wild card) patterns with parenthesized sub-expressions. The \$1 is a scripting language variable which evaluates to the first parenthesized sub-expression in the active target pattern. The \$req refers to the single prerequisite. The \$CC and \$CFLAGS are just predefined variables in the scripting language.

This kind of pattern-based target specification increases the programmer's or user's flexibility and power in constructing rules. For example, if \$sdir is a private

directory containing the master copy of some files, and \$tdir is a public directory meant to contain updated copies of those files, one can specify this dependency with:

```
$tdir/(*): $sdir/$1  
exec cp -p $req $trg
```

5 Thus, a generic rule can be provided for updating any file in \$tdir from the corresponding file in \$sdir.

Regular expression patterns are more powerful than glob patterns, and may be needed in some cases. For example, suppose executable scripts (without suffixes) are to be generated from corresponding scripts with a ".src" suffice by a *sed* path 10 substitution. This rule might be specified using regular expressions as follows:

```
set patternType regexp  
bin/(^[^.]*$): bin/$1.src  
exec sed -e "s?actual_path?template_path?g" $reg >$trg
```

This rule says that any concrete target in the bin directory which has no period 15 in the name may be generated from the corresponding file in the bin directory with a ".src" extension via the supplied *sed* command.

When target patterns are used in *upd* control files, there are occasions when it is helpful to be able to specify the dependency and update rules associated with given file separately.

20 When a particular concrete file is needed, *upd* preferably uses the prerequisites from the first matching target pattern with an associated prerequisite list, and the update rule from the first matching target pattern with an attached update rule.

Consider the following *upd* control file:

```
inst/bin/(*): master/bin/$1  
inst/bin/(*).e"  
exec sed -e "s?actual_path?template_path?g" $req >$trg  
inst/bin/(*):  
exec cp -p $req $trg
```

SPS

The first rule says that any file in the inst/bin directory depends on the corresponding file in the master/bin directory. The second rule says that any needed file in the inst/bin with a ".e" suffix can be created via a set substitution on its prerequisite. The third rule says that any needed file in the inst/bin should be simple a copy of its prerequisite. Because it comes after the ".e" rule, however, it does not get triggered for inst/bin/*.e files. It is only triggered by inst/bin files without a ".e" suffix.

The value of \$req for the second two rules is determined from the first rule, which supplies the prerequisite list for every file in the inst/bin.

To reiterate, when searching for a dependency list or update rule to apply to a needed concrete target, update checks that target against the target patterns in the order in which those patterns are specified in the *upd*t control file. For this reason, rules with more-specific target patterns should precede rules with less-specific target patterns.

The *upd*t utility treats directory targets in a special manner, to support automatic recursive processing of their contents: Directory targets are always considered to be out-of-date. Thus, the rule associated with a directory target is always triggered when that directory is determined to be needed, regardless of the directory's actual status.

How does *upd*t know whether it is dealing with a directory target or a file target? If the target entity does not exist, it does not matter. The target is always out-of-date in this case. If the target entity does exist, *upd*t simply checks to see whether it is a directory or file, and if it is a directory, it considers it out-of-date regardless of its time stamp.

Another way *upd*t supports automated processing of directory contents is by providing a process procedure for the user to call. The process procedure recursively invokes the *upd*t engine on all its arguments.

To illustrate how *upd*t's directory processing features might be used, suppose there is a target directory, \$tdir, and the goal is to update all the files and

subdirectories in it from a corresponding source directory, \$sdir. The last *upd* control file rule in this case might look like:

```
$tdir/(*): $sdir/$1
      if [file isdirectory $req] then {
          5       file mkdir $trg
          foreach f [glob -nocomplain $req/* $req/.*] {
              if {$f != "." && $f != ".."} then {
                  process $tdir/[file tail $req]
              }
          }
          10      } else {
                  exec cp -p $req $trg
              }
      }
```

The dependency establishes that every file or directory in the \$tdir depends on
15 the corresponding entity in \$sdir. The update rule treats directories differently from
files. Note that the update rule checks the type of \$req, not \$trg, since \$trg may not
exist yet.

For targets dependent on \$req files (as opposed to dependant on \$req
directories), the user copies the prerequisite (file in \$sdir) to the target (corresponding
20 file in \$tdir).

For targets dependent on \$req subdirectories, the user first makes the
corresponding \$trg subdirectory, and then recursively invokes the *upd* engine on all
of the entities in the \$req directory (except '.' and '..'), using the three process
command. This single rule states that \$tdir is a recursive copy of the \$sdir, but when
25 it is triggered, only out-of-date files in \$sdir will be copied to \$tdir.

Suppose, for example, that for certain files (e.g files with a '.e' suffix) the
user did not want a simple copy, the user wanted a *sed* substitution and then a copy.
The user could then simply precede the above rule with:

```
$tdir/*.e:
```

exec sed -e "s?actual_path?template_path?g" \$req >\$trg

which would override the first update rule in the case of files with '.e' extensions. The prerequisite would still be determined from the more general dependency/update rule already described.

5 The method of the present invention may be embodied as a computer program product stored on a program storage device. This program storage device may be devised, made and used as a component of a machine utilizing optics, magnetic properties and/or electronics to perform the method steps of the present invention. Program storage devices include, but are not limited to, magnetic disks or diskettes,
10 magnetic tapes, optical disks, Read Only Memory (ROM), floppy disks, semiconductor chips and the like. A computer readable program code means in known source code may be employed to convert the methods described below for use on a computer. The computer program or software incorporating the process steps and instructions described further below may be stored in any conventional computer,
15 for example, that shown in Fig. 4. Computer 150 incorporates a program storage device 152 and a microprocessor 154. Installed on the program storage device 152 is the program code incorporating the method of the present invention, as well as any database information for the mask pattern of a feature to be created on the semiconductor substrate and the lithographic process window variations.

20 While the present invention has been particularly described, in conjunction with a specific preferred embodiment, it is evident that many alternatives, modifications and variations will be apparent to those skilled in the art in light of the foregoing description. It is therefore contemplated that the appended claims will embrace any such alternatives, modifications and variations as falling within the true scope and spirit of the present invention.
25

Thus, having described the invention, what is claimed is: